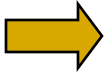
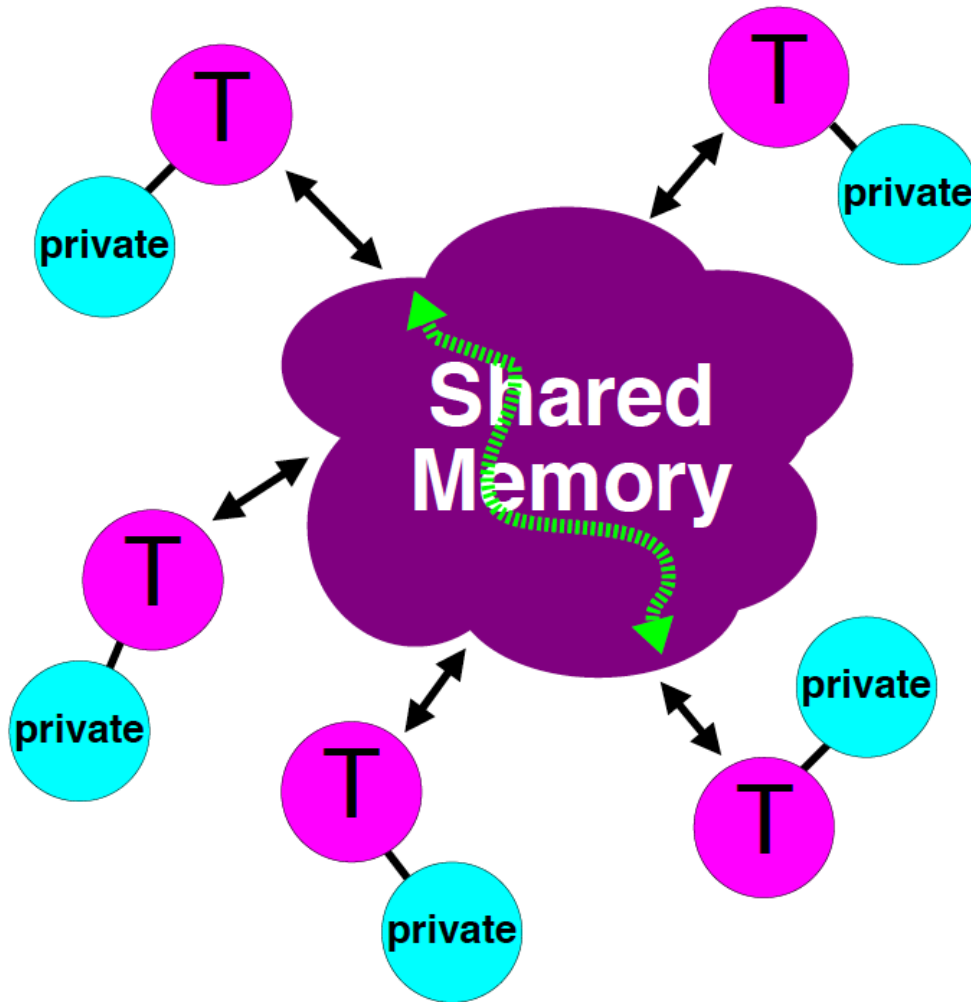


Agenda

- ❑ What is OpenMP?
- ❑ The core elements of OpenMP
 - ❑ Parallel regions
 - ❑ Work-sharing constructs
 - ❑ Synchronization
 -  ❑ Managing the data environment
 - ❑ The runtime library and environment variables
 - ❑ Tasks
- ❑ OpenMP usage
 - ❑ An example

OpenMP Memory Model



- ✓ All threads have access to the same, globally shared, memory
- ✓ Data can be shared or private
- ✓ Shared data is accessible by all threads
- ✓ Private data can only be accessed by the thread that owns it
- ✓ Data transfer is transparent to the programmer
- ✓ Synchronization takes place, but it is mostly implicit

OpenMP Data Environment

- ❑ Most variables are shared by default
- ❑ Global variables are SHARED among threads
 - ❑ Fortran: COMMON blocks, SAVE variables, MODULE variables
 - ❑ C: File scope variables, static
- ❑ But not everything is shared by default...
 - ❑ Stack variables in sub-programs called from parallel regions are PRIVATE
 - ❑ Automatic variables defined inside the parallel region are PRIVATE.
- ❑ The default status can be modified with:
 - ❑ DEFAULT (PRIVATE | SHARED | NONE)

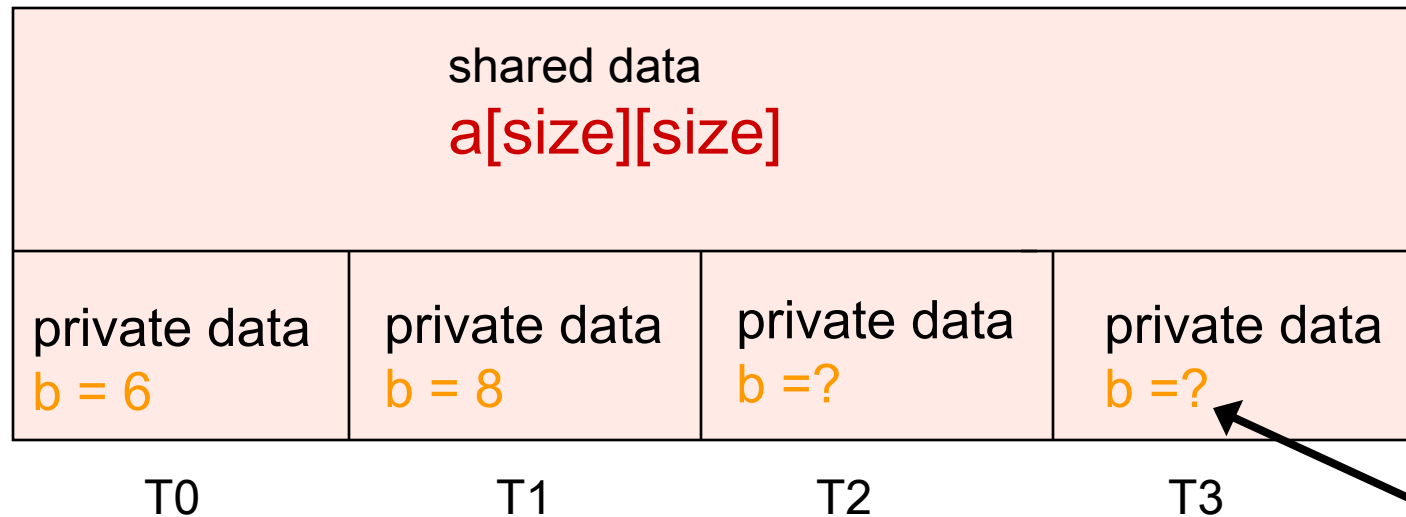
All data clauses apply to parallel regions, tasks and work-sharing constructs except “shared” which does not apply to work-sharing constructs .

About Storage Association

- ❑ Private variables are *undefined* on entry and exit of the parallel region
- ❑ A private variable within a parallel region has *no* storage association with the same variable outside of the region
- ❑ Use the **firstprivate** and **lastprivate** clauses to override this behavior
- ❑ We illustrate these concepts with an example

OpenMP Data Environment

```
double a[size][size], b=4;  
#pragma omp parallel private (b)  
{ .... }
```



Private variable `b`
becomes undefined on
exit from region

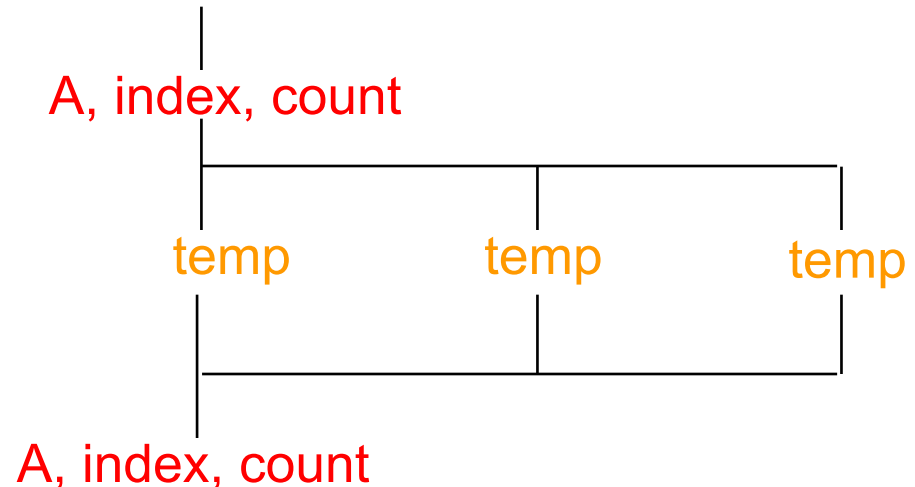
OpenMP Data Environment

```
program sort
common /input/ A(10)
integer index(10)
C$OMP PARALLEL
  call work (index)
C$OMP END PARALLEL
print*, index(1)
```

```
subroutine work (index)
common /input/ A(10)
integer index(*)
real temp(10)
integer count
  save count
  .....
```

A, index and count are shared by all threads.

temp is local to each thread



OpenMP Private Clause

- ❑ **private**(var) creates a local copy of var for each thread.
 - ❑ The value is *uninitialized*
 - ❑ Private copy is *not storage-associated* with the original
 - ❑ Parallel region does not modify original variable

```
IS = 0
C$OMP PARALLEL DO PRIVATE(IS)
DO 1, 1000
  IS = IS + J
END DO
C$OMP END PARALLEL DO
print *, IS
```

IS was not
initialized

IS here is not
storage-associated
with the private
variable with the
same name

(In)Visibility of Private Data

```
#pragma omp parallel private(x) shared(p0, p1)
```

Thread 0

X = ...;

P0 = &x;

/* references in the following line are **not** allowed */
... *p1 ...

Thread 1

X = ...;

P1 = &x;

... *p0 ...

- ❑ You can not reference another's threads private variables ... even if you have a shared pointer between the two threads.

The Firstprivate And Lastprivate Clauses

firstprivate (list)

- *All variables in the list are initialized with the value the original object had before entering the parallel construct*

lastprivate (list)

- *The thread that executes the sequentially last iteration or section updates the value of the objects in the list*

Firstprivate Clause

- **firstprivate** is a special case of private.
 - Initializes each private copy with the corresponding value from the master thread.

```
IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
  DO 20 J=1,1000
    IS = IS + J
  20 CONTINUE
C$OMP END PARALLEL DO
print *, IS
```



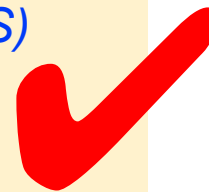
Each thread gets its own IS with an initial value of 0

The value of IS here is not influenced by the computation in the parallel region

Lastprivate Clause

- ❑ **Lastprivate** passes the value of a private variable from the last iteration to the variable of the master thread

```
IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
C$OMP& LASTPRIVATE(IS)
  DO 20 J=1,1000
    IS = IS + J
  20 CONTINUE
C$OMP END PARALLEL DO
print *, IS
```



Each thread gets its own IS with an initial value of 0

IS is defined as its value at the last iteration (i.e. for J=1000)

A Data Environment Checkup

- ❑ Consider this example of PRIVATE and FIRSTPRIVATE

```
C      variables A,B, and C = 1
C$OMP PARALLEL PRIVATE(B)
C$OMP& FIRSTPRIVATE(C)
```

- ❑ Are A,B,C local to each thread or shared inside the parallel region?
- ❑ What are their initial values inside and after the parallel region?

Inside this parallel region ...

- “A” is shared by all threads; equals 1
- “B” and “C” are local to each thread.
 - B’s initial value is undefined
 - C’s initial value equals 1

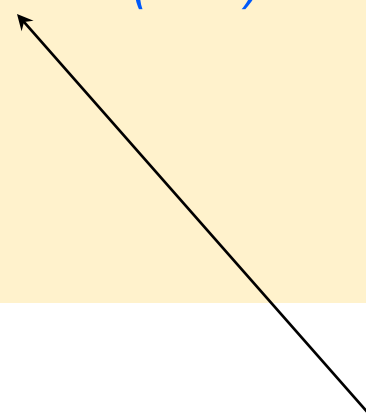
Outside this parallel region ...

- A has value from parallel region. The values of “B” and “C” are not influenced by code inside region.

OpenMP Reduction

- ❑ If it's the sum of all J values that you need, there is a way to do that too.
- ❑ We have already seen how

```
IS = 0
C$OMP PARALLEL DO REDUCTION(+:IS)
  DO 1000 J=1,1000
    IS = IS + J
  1000 CONTINUE
  print *, IS
```



Result variable is shared by default

OpenMP Reduction

reduction (operator: list)

C/C++

reduction ([operator | intrinsic]) : list)

Fortran

- ❑ An accumulation operation across threads
- ❑ Inside a parallel or work-sharing construct:
 - ❑ A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
 - ❑ Compiler finds standard reduction expressions containing “op” and uses them to update the local copy.
 - ❑ Local copies are reduced into a single value and combined with the original global value.
- ❑ The variables in “list” must be shared in the enclosing parallel region.

Reduction Operands/Initial Values

- ❑ Associative operands used with reduction
- ❑ Initial values are the ones that make sense mathematically

Operand	Initial value
+	0
*	1
-	0
.AND.	All 1' s

Operand	Initial value
.OR.	0
MAX	1
MIN	0
//	All 1' s

The Default Clause

default (none | shared)

C/C++

default (none | shared | private | threadprivate)

Fortran

none

- *No implicit defaults; have to scope all variables explicitly*

shared

- *All variables are shared*
- *The default in absence of an explicit "default" clause*

private

- *All variables are private to the thread*
- *Includes common block data, unless THREADPRIVATE*

firstprivate

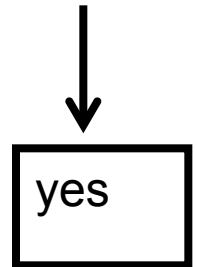
- *All variables are private to the thread; pre-initialized*

Default Clause Example

```
    itotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
    np = omp_get_num_threads()
    each = itotal/np
    .....
C$OMP END PARALLEL
```

Are these
two codes
equivalent?

```
    itotal = 1000
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal)
    np = omp_get_num_threads()
    each = itotal/np
    .....
C$OMP END PARALLEL
```



Example - Parallelizing Bulky Loops

```
for (i=0; i<n; i++) /* Parallel loop */
{
    a = ...
    b = ... a ..
    c[i] = ....
    .....
    for (j=0; j<m; j++)
    {
        <a lot more data and work in this loop>
    }
    .....
}
```

Step 1: “Outlining”

```
for (int i=0; i<n; i++) /* Parallel loop */  
{  
    (void) FuncPar(i,m,c,...)  
}
```

- ✓ *Still a sequential program*
- ✓ *Should behave identically*
- ✓ *Easy to test for correctness*
- ✓ *Simplifies the parallelization*

```
void FuncPar(i,m,c,...)  
{  
    float a, b; /* Private data */  
    int    j;  
    a = ...  
    b = ... a ..  
    c[i] = ....  
        .....  
    for (j=0; j<m; j++)  
    {  
        <a lot more work in this loop>  
    }  
        .....  
}
```

Step 2: Parallelize

```
#pragma omp parallel for private(..) shared(..)
```

```
for (int i=0; i<n; i++) /* Parallel loop */  
{  
    (void) FuncPar(i,m,c,...)  
} /*-- End of parallel for --*/
```

✓ **Minimal scoping required**

✓ **Less error prone**

```
void FuncPar(i,m,c,...)  
{  
    float a, b; /* Private data */  
    int    j;  
    a = ...  
    b = ... a ..  
    c[i] = ....  
    .....  
    for (j=0; j<m; j++)  
    {  
        <a lot more work in this loop>  
    }  
    .....  
}
```

OpenMP Threadprivate

- ❑ Makes global data private to a thread and persistent, *thus crossing parallel region boundary*
 - ❑ Fortran: COMMON blocks
 - ❑ C: File scope and static variables
- ❑ Different from making them PRIVATE
 - ❑ With PRIVATE, global variables are masked.
 - ❑ **THREADPRIVATE** preserves global scope within each thread
- ❑ Threadprivate variables can be initialized using COPYIN or by using DATA statements.
- ❑ Some limitations on use of threadprivate
 - ❑ Consult specification before using this feature

A Threadprivate Example

- Consider two different routines called within a parallel region.

```
subroutine poo
parameter (N=1000)
common/buf/A(N),B(N)
!$OMP THREADPRIVATE(/buf/)
do i=1, N
  B(i)= const* A(i)
end do
return
end
```

```
subroutine bar
parameter (N=1000)
common/buf/A(N),B(N)
!$OMP THREADPRIVATE(/buf/)
do i=1, N
  A(i) = sqrt(B(i))
end do
return
end
```

Because of the **threadprivate** construct, each thread executing these routines has its own copy of the common block /buf/.

Values of threadprivate are persistent across parallel regions

The Copyin Clause

copyin (list)

- ❑ Applies to THREADPRIVATE data only
- ❑ At the start of the parallel region, data of the master thread is copied to the thread private copies

Example:


```
common /cblock/velocity
common /fields/xfield, yfield, zfield

! create thread private common blocks

!$omp threadprivate (/cblock/, /fields/)

!$omp parallel          &
!$omp default (private) &
!$omp copyin ( /cblock/, zfield )
```

Data now
available to
threads



Copyprivate

- ❑ Used with a single region to broadcast values of private variables from one member of a team to the rest of the team.

```
#include <omp.h>
void input_parameters (int, int); // fetch values of input parameters
void do_work(int, int);

void main()
{
    int Nsize, choice;

    #pragma omp parallel private (Nsize, choice)
    {
        . . . . .
        #pragma omp single copyprivate (Nsize, choice)
            input_parameters (Nsize, choice);

        do_work(Nsize, choice);
    }
}
```


C++ And Threadprivate

- ❑ OpenMP 3.0 clarified where/how threadprivate objects are constructed and destructed
- ❑ Allows C++ static class members to be threadprivate

```
class T {  
    public:  
    static int i;  
    #pragma omp threadprivate(i)  
    ...  
};
```

Agenda

- ❑ What is OpenMP?
- ❑ The core elements of OpenMP
 - ❑ Parallel regions
 - ❑ Work-sharing constructs
 - ❑ Synchronization
 - ❑ Managing the data environment
 - ➔ ❑ The runtime library and environment variables
 - ❑ Tasks
- ❑ OpenMP usage
 - ❑ An example

OpenMP Runtime Functions

- ❑ OpenMP provides a set of runtime functions
 - ❑ They all start with “omp_”
- ❑ These functions can be used to:
 - ❑ Query for a specific feature, value or setting
 - ❑ E.g. what is my thread ID?
 - ❑ Change a setting
 - ❑ E.g. to change the number of threads in next parallel region
 - ❑ A special category consists of the locking functions

C/C++ : Need to include file <omp.h>

Fortran : Add “use omp_lib” or include file “omp_lib.h”

OpenMP Library Routines

- ❑ Modify/Check the number of threads
 - ❑ `omp_set_num_threads()`, `omp_get_num_threads()`,
`omp_get_thread_num()`, `omp_get_max_threads()`
- ❑ Are we in a parallel region?
 - ❑ `omp_in_parallel()`
- ❑ How many processors in the system?
 - ❑ `omp_num_procs()`

OpenMP Library Routines

- ❑ To use a known, fixed number of threads used in a program, (1) tell the system that you don't want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.

```
#include <omp.h>
```

```
void main()
```

```
{ int num_threads;
```

```
    omp_set_dynamic( 0 );
```

```
    omp_set_num_threads( omp_num_procs() );
```

```
#pragma omp parallel
```

```
{ int id=omp_get_thread_num();
```

```
#pragma omp single
```

```
    num_threads = omp_get_num_threads();
```

```
    do_lots_of_stuff(id);
```

```
}
```

```
}
```

Disable dynamic adjustment of the number of threads.

E.g. Request as many threads as you have processors.

Protect this op since memory stores are not atomic

Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.

OpenMP Runtime Functions

Name

`omp_set_num_threads`

`omp_get_num_threads`

`omp_get_max_threads`

`omp_get_thread_num`

`omp_get_num_procs`

`omp_in_parallel`

`omp_set_dynamic`

`omp_get_dynamic`

`omp_set_nested`

`omp_get_nested`

`omp_get_wtime`

`omp_get_wtick`

Functionality

Set number of threads

Number of threads in team

Max num of threads for parallel region

Get thread ID

Maximum number of processors

Check whether in parallel region

Activate dynamic thread adjustment

(but implementation is free to ignore this)

Check for dynamic thread adjustment

Activate nested parallelism

(but implementation is free to ignore this)

Check for nested parallelism

Returns wall clock time

Number of seconds between clock ticks

Schedule-Related Functions

- ❑ Makes `schedule(runtime)` more general
- ❑ Can set/get schedule with library routines:

```
omp_set_schedule()  
omp_get_schedule()
```

- ❑ Implementations are also allowed to add their own schedule kinds

Nested Parallelism

- ❑ Allows parallel regions to be contained in each other
 - ❑ Often accomplished by having parallel regions in different functions
- ❑ Required: `OMP_NESTED=true` or `omp_set_nested(1)`
 - ❑ Else the inner parallel region will be executed by a team of one thread (may happen anyway)
- ❑ Total number of threads created is the *product* of the number of threads in the teams at each level
 - ❑ Use `omp_set_num_thread(n)` or the `num_threads()` clause

Multiple levels of nesting team sizes can be defined via the

- ❑ `OMP_NUM_THREADS` environment variable
- ❑ `setenv OMP_NUM_THREADS 4,2`

Nested Parallelism Support

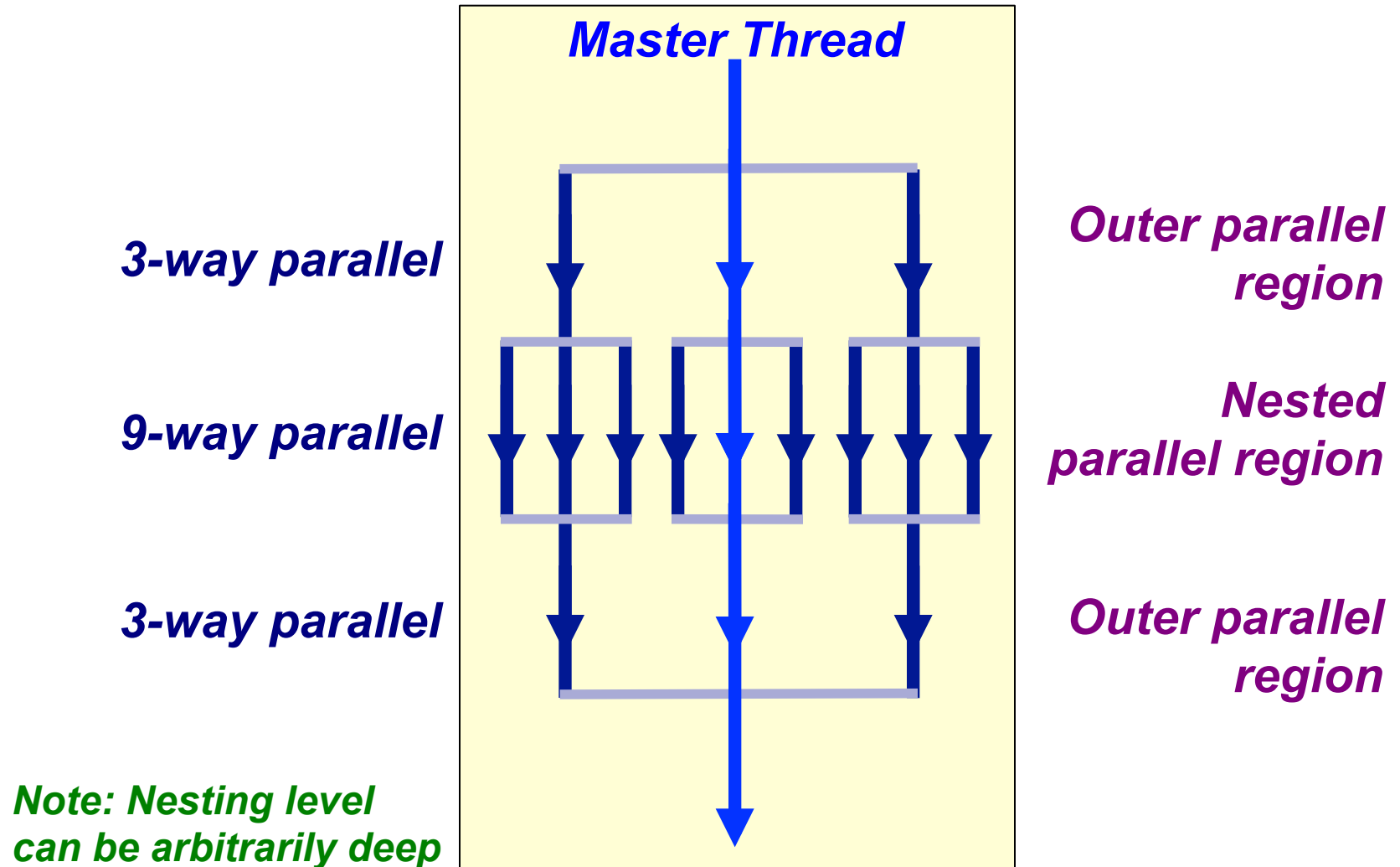
- ❑ Environment variable and runtime routines to set/get the maximum number of nested active parallel regions

```
OMP_MAX_ACTIVE_LEVELS  
omp_set_max_active_levels()  
omp_get_max_active_levels()
```

- ❑ Environment variable and runtime routine to set/get the maximum number of OpenMP threads available to the program

```
OMP_THREAD_LIMIT  
omp_get_thread_limit()
```

Nested Parallelism



Nested Parallelism Support

- ❑ Settings can apply to level
 - ❑ Allow, for example, calling `omp_set_num_threads()` inside a parallel region to control the team size for next level of parallelism
 - ❑ Library routines to determine
 - ❑ Depth of nesting
 - ❑ `omp_get_level()`
 - ❑ `omp_get_active_level()`
 - ❑ IDs of parent/grandparent etc. threads
 - ❑ `omp_get_ancestor_thread_num(level)`
 - ❑ Team sizes of parent/grandparent etc. teams
 - ❑ `omp_get_team_size(level)`
-

OpenMP Locking Routines

- ❑ Locks provide greater flexibility than critical regions and atomic updates:
 - ❑ Possible to implement asynchronous behavior
 - ❑ Not block structured
- ❑ The so-called lock variable is a special variable:
 - ❑ C/C++: type `omp_lock_t` and `omp_nest_lock_t` for nested locks
 - ❑ Fortran: type `INTEGER` and of a `KIND` large enough to hold an address
- ❑ Lock variables are manipulated through the API only
- ❑ Using a lock variable without appropriate initialization is illegal, and behavior is undefined

Locking Routines

- ❑ Simple locks: may not be locked if already in a locked state
- ❑ Nestable locks: may be locked multiple times by the same thread before being unlocked
- ❑ The interface for functions dealing with nested locks is similar (but using nestable lock variables):

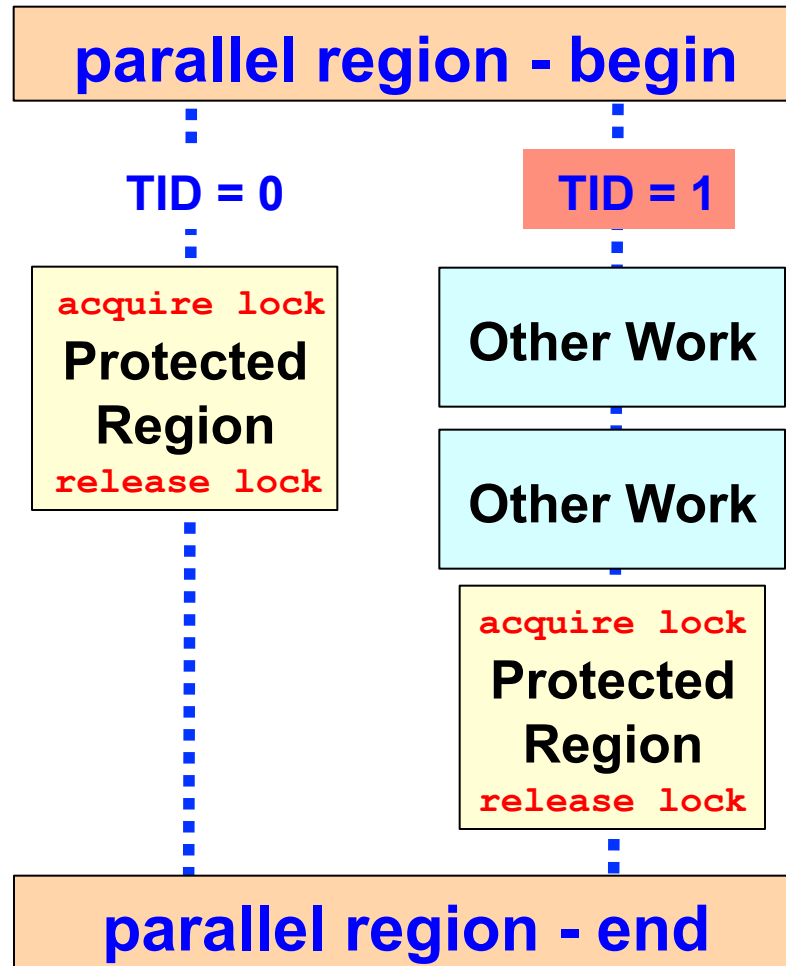
Simple locks

```
omp_init_lock  
omp_destroy_lock  
omp_set_lock  
omp_unset_lock  
omp_test_lock
```

Nestable locks

```
omp_init_nest_lock  
omp_destroy_nest_lock  
omp_set_nest_lock  
omp_unset_nest_lock  
omp_test_nest_lock
```

OpenMP Locking Example



- ◆ *The protected region contains the update of a shared variable*
- ◆ *One thread acquires the lock and performs the update*
- ◆ *Meanwhile, the other thread performs some other work*
- ◆ *When the lock is released again, the other thread performs the update*

Locking Example - The Code

```
Program Locks
....
Call omp_init_lock (LCK)

!$omp parallel shared(LCK)

    Do While ( omp_test_lock (LCK) .EQV. .FALSE. )
        Call Do_Something_Else()
    End Do

    Call Do_Work()

    Call omp_unset_lock (LCK)

!$omp end parallel

    Call omp_destroy_lock (LCK)

Stop
End
```

Initialize lock variable

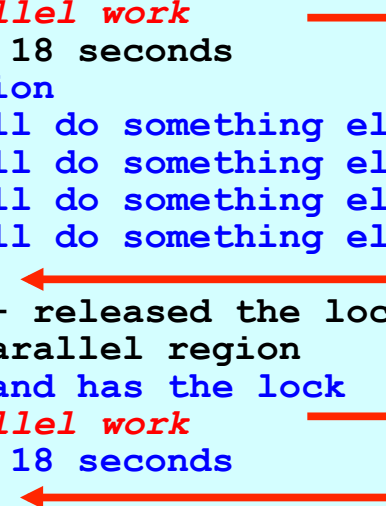
Check availability of lock
(also sets the lock)

Release lock again

Remove lock association

Example Output Using 2 Threads

```
TID: 1 at 09:07:27 => entered parallel region
TID: 1 at 09:07:27 => done with WAIT loop and has the lock
TID: 1 at 09:07:27 => ready to do the parallel work
TID: 1 at 09:07:27 => this will take about 18 seconds
TID: 0 at 09:07:27 => entered parallel region
TID: 0 at 09:07:27 => WAIT for lock - will do something else for 5 seconds
TID: 0 at 09:07:32 => WAIT for lock - will do something else for 5 seconds
TID: 0 at 09:07:37 => WAIT for lock - will do something else for 5 seconds
TID: 0 at 09:07:42 => WAIT for lock - will do something else for 5 seconds
TID: 1 at 09:07:45 => done with my work
TID: 1 at 09:07:45 => done with work loop - released the lock
TID: 1 at 09:07:45 => ready to leave the parallel region
TID: 0 at 09:07:47 => done with WAIT loop and has the lock
TID: 0 at 09:07:47 => ready to do the parallel work
TID: 0 at 09:07:47 => this will take about 18 seconds
TID: 0 at 09:08:05 => done with my work
TID: 0 at 09:08:05 => done with work loop - released the lock
TID: 0 at 09:08:05 => ready to leave the parallel region
Done at 09:08:05 - value of SUM is 1100
```



Used to check the answer

Note: program was instrumented to get this information

OpenMP Environment Variables/1

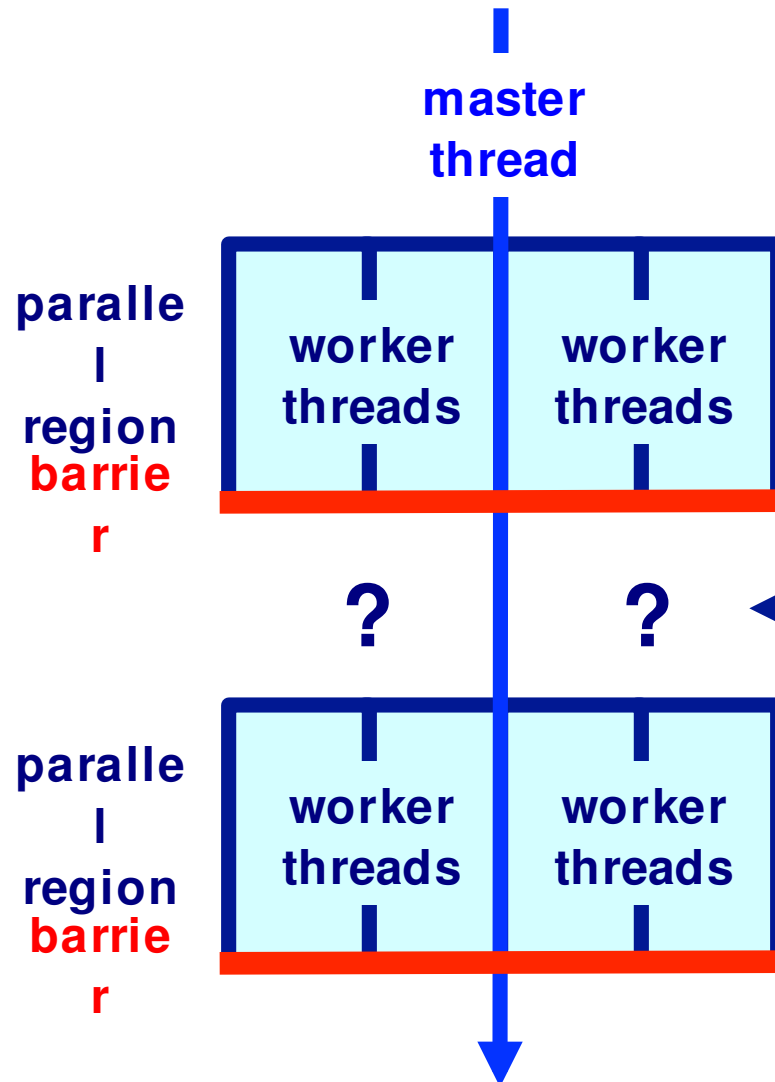
OpenMP Environment Variable	IBM XL Compilers
OMP_NUM_THREADS	64 (BG/Q) number of available processors (other systems)
OMP_SCHEDULE “schedule,[chunk]”	auto
OMP_DYNAMIC {TRUE FALSE}	FALSE
OMP_NESTED {TRUE FALSE}	FALSE
OMP_STACKSIZE “size [B K M G]”	256 MB (32 bit) up to available resource (64 bit)
OMP_WAIT_POLICY [ACTIVE PASSIVE]	PASSIVE
OMP_MAX_ACTIVE_LEVELS	5

- ❑ ***Be careful when relying on defaults (because they are compiler dependent)***

OpenMP Environment Variables/2

OpenMP Environment Variable	Default Oracle Solaris Studio
OMP_THREAD_LIMIT	64 (BG/Q) max(OMP_NUM_THREADS, number of available processors) (other systems)
OMP_PROC_BIND {TRUE FALSE}	TRUE (BG/Q, FALSE is ignored) FALSE (other systems)

Implementing The Fork-Join Model



*Use the
OMP_WAIT_POLICY
environment variable to
control the behavior of
idle threads
Values: ACTIVE, PASSIVE*

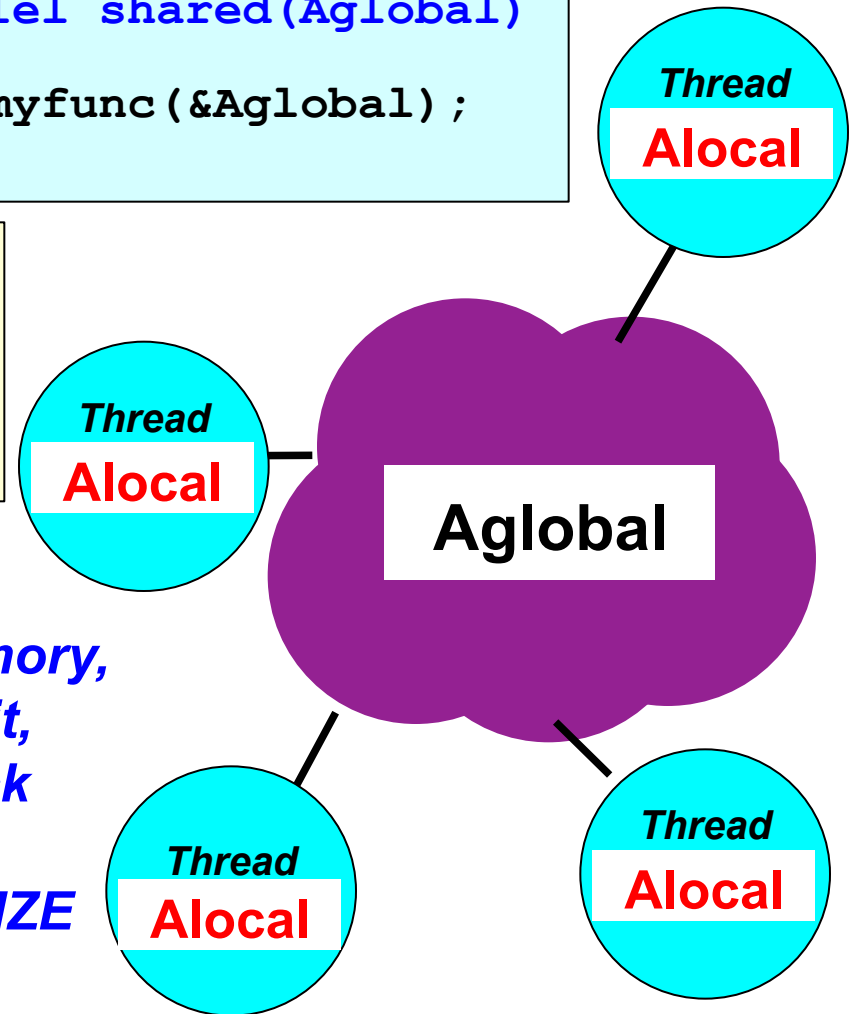
About The Stack

```
#omp parallel shared(Aglobal)
{
    (void) myfunc (&Aglobal) ;
}
```

```
void myfunc(float *Aglobal)
{
    int Alocal;
    .....
}
```

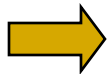
*Variable **Alocal** is in private memory, managed by the thread owning it, and stored on the so-called stack*

*Set stacksize via **OMP_STACKSIZE** environment variable*



Agenda

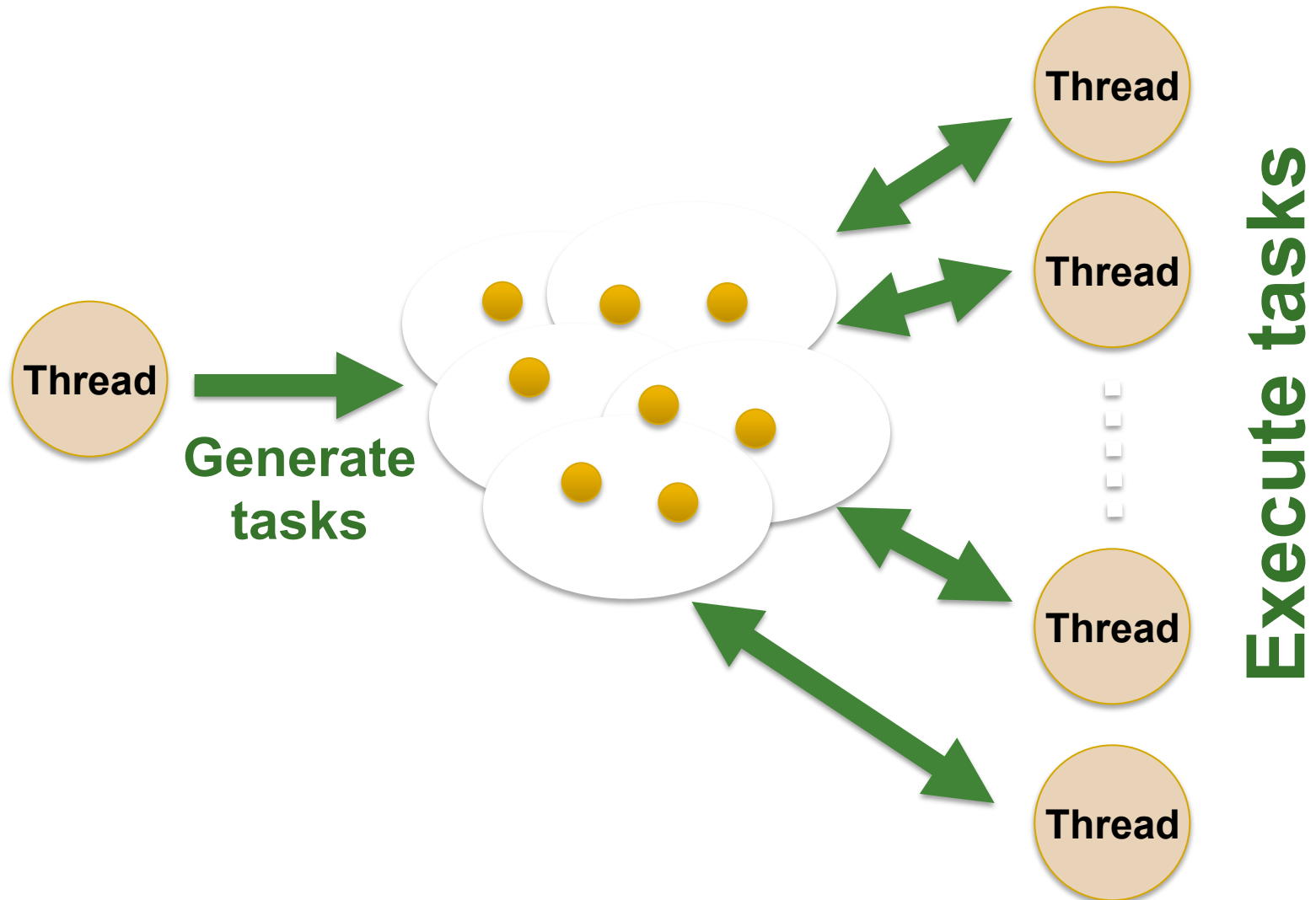
- ❑ What is OpenMP?
- ❑ The core elements of OpenMP
 - ❑ Parallel regions
 - ❑ Work-sharing constructs
 - ❑ Synchronization
 - ❑ Managing the data environment
 - ❑ The runtime library and environment variables
 - ❑ Tasks
- ❑ OpenMP usage
 - ❑ An example



Tasking In OpenMP

- ❑ Tasking was introduced in OpenMP 3.0
- ❑ Until then it was impossible to efficiently implement certain types of parallelism
 - ❑ Recursive algorithms
 - ❑ Linked lists, ...
- ❑ The initial functionality was very simple by design
 - ❑ The idea was (and still) is to augment tasking as we collectively gain more insight and experience

The Tasking Concept In OpenMP



The Tasking Construct

Define a task:

```
#pragma omp task
```

```
!$omp task
```

- ❑ A **task** is a specific instance of executable code and its data environment
- ❑ A task is generated when a thread encounters a task construct or a parallel construct. Comprised of a task region and data environment.
- ❑ A task region consists of all code encountered during the execution of a task.
- ❑ The data environment consists of all the variables associated with the execution of a given task. It is constructed from the data environment of the generating task at the time the task is generated.

Tasking - Who Does What And When ?

- ❑ Assumption: all tasks can execute independently
- ❑ When any thread encounters a **task construct**, a new task is generated
 - ❑ Tasks can be nested (but not for the faint of heart)
- ❑ Execution of a generated task is carried out by one of the threads in the current team
 - ❑ This is subject to the thread's availability and thus could be immediate or deferred until later
- ❑ Completion of the task can be guaranteed using a **task synchronization** construct
 - ❑ a **taskwait** or a **barrier** construct

Task Completion

Explicit wait on the completion of child tasks:

```
int fib(int n) {  
    int x, y;  
    if (n < 2) return n;  
    else {  
        #pragma omp task shared(x)  
        x = fib(n-1);  
        #pragma omp task shared(y)  
        y = fib(n-2);  
        #pragma omp taskwait  
        return x + y;  
    }  
}
```

```
#pragma omp taskwait
```

```
!$omp taskwait
```

Does not include descendents of child tasks

Tasking Example

```
int main(int argc, char *argv[]) {  
    #pragma omp parallel  
    {  
        #pragma omp single  
        {  
            printf("A ");  
            #pragma omp task  
            {printf("race ");}  
            #pragma omp task  
            {printf("car ");}  
            printf("is fun to watch ");  
        }  
    } // End of parallel region  
  
    printf("\n");  
    return(0);  
}
```

*What will this program print using
2 threads ?*

Tasking Example

```
$ cc -xopenmp -fast hello.c  
$ export OMP_NUM_THREADS=2  
$ ./a.out
```

```
A is fun to watch race car  
$ ./a.out
```

```
A is fun to watch race car  
$ ./a.out
```

```
A is fun to watch car race  
$
```

Tasking Example

```
int main(int argc, char *argv[]) {  
    #pragma omp parallel  
    {  
        #pragma omp single  
        {  
            printf("A ");  
            #pragma omp task  
            {printf("car ");}  
            #pragma omp task  
            {printf("race ");}  
            #pragma omp taskwait  
            printf("is fun to watch ");  
        }  
    } // End of parallel region  
  
    printf("\n"); return(0);  
}
```

***What will this program
print using 2 threads ?***

Tasking Example

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
$
A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch
$
```

Tasks are executed first now

Clauses On The Task Directive

if(*scalar-expression*)

if false, create an undeferred task:
encountering thread must suspend
the encountering task region, immediately
execute the current task region until it is
completed. Helps avoid small tasks.
any thread can resume after suspension

untied

default(shared | none)

private(*list*)

firstprivate(*list*)

shared(*list*)

final(*scalar-expression*)

mergeable

if true, the generated task is a final task
if the task is an undeferred task or an
included task, the implementation may
generate a merged task

Data Scoping in Tasks

- ❑ Static and global variables are shared
- ❑ Automatic storage (local) variables are private
- ❑ Variables are firstprivate unless shared in the enclosing context

```
int a;
void foo()
{
    int b, c;

    #pragma omp parallel private(b)
    {
        int d;
        #pragma omp task
        {
            int e;
            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d: firstprivate
            // Scope of e: private
        }
    }
}
```


Task Scheduling Points In OpenMP

- ❑ Whenever a thread reaches a **task scheduling point**, it may suspend the current task in order to execute a different task bound to the current team
- ❑ Task scheduling points are implied at:
 - ❑ The point immediately following the generation of an explicit task
 - ❑ After the last instruction of a task region
 - ❑ In taskwait and taskyield regions
 - ❑ In implicit and explicit barrier regions
- ❑ The implementation may insert task scheduling points in untied tasks
- ❑ The user may define additional scheduling points

Tied and Untied Tasks

- ❑ Default: Tasks are tied to the thread that first executes them
 - ❑ Tasks created with the **untied** clause are never tied to a thread
 - ❑ Take care with some constructs, e.g. thread ids, locks
- ❑ This affects execution behavior after a *task switch* at a task scheduling point
- ❑ If the suspended task region is for a tied task, the initially assigned thread resumes execution of the suspended task subsequently
 - ❑ If it is untied, any thread may resume its execution

Example: A Linked List

```
.....  
while(my_pointer) {  
  
    (void) do_independent_work (my_pointer);  
    my_pointer = my_pointer->next ;  
} // End of while loop  
.....
```

***Hard to do before OpenMP 3.0:
First count number of iterations, then
convert while loop to for loop***

Example: A Linked List with Tasking

```
my_pointer = listhead;
#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(my_pointer) {
            #pragma omp task firstprivate(my_pointer)
            {
                (void) do_independent_work (my_pointer);
            }
            my_pointer = my_pointer->next ;
        }
    } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```

OpenMP Task is specified here
(executed in parallel)



Taskyield

```
#pragma omp taskyield
```

```
!$omp taskyield
```

- ❑ The taskyield directive specifies that the current task can be suspended in favor or execution of a different task
- ❑ Hint to the runtime

```
#include <omp.h>
void something_useful();
void something_critical();
void foo(omp_lock_t * lock, int n)
{
    for(int i = 0; i < n; i++)
        #pragma omp task
        {
            something_useful();
            while( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
}
```

The waiting task may be suspended here so that the executing thread can perform other work.

Final clause

```
#pragma omp task final(expr)
```

```
!$omp task final(expr)
```

- ❑ For recursive problems that perform task decomposition
 - ❑ stop task creation at a certain depth exposes
 - ❑ enough parallelism while reducing overhead.
- ❑ Warning: Merging the data environment may have side-effects

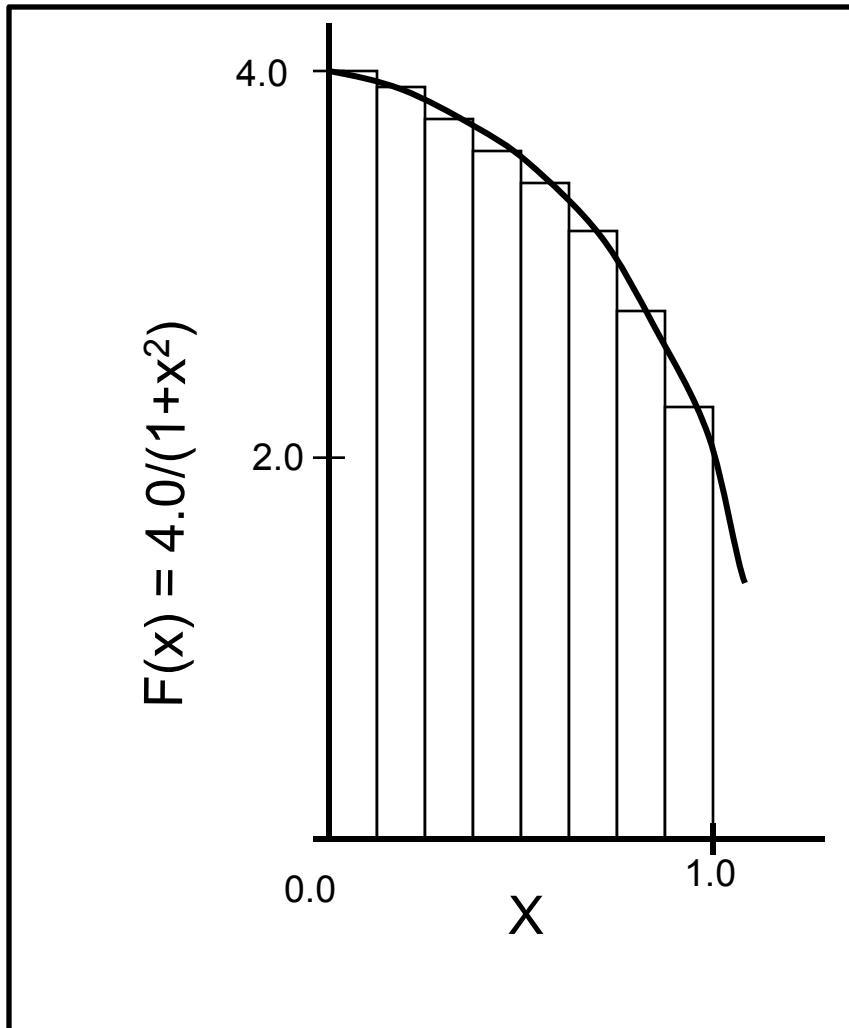
```
void foo(bool arg)
{
    int i = 3;
    #pragma omp task final(arg) firstprivate(i)
        i++;
        printf("%d\n", i); // will print 3 or 4 depending on arg
}
```

Agenda

- ❑ What is OpenMP?
- ❑ The core elements of OpenMP
 - ❑ Parallel regions
 - ❑ Work-sharing constructs
 - ❑ Synchronization
 - ❑ Managing the data environment
 - ❑ The runtime library and environment variables
 - ❑ Tasks
- ➡ ❑ OpenMP usage
 - ❑ An example

Let's pause for a quick recap by example:

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i .

PI Program: an example

```
static long num_steps = 100000;
double step;
void main ()
{
    int i;    double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    for (i=0;i<= num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

OpenMP recap:

Parallel Region

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{
    int i, id, nthreads; double x, pi, sum[NUM_THREADS];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (i, id, x)
{
    id = omp_get_thread_num();
#pragma omp single
    nthreads = omp_get_num_threads()
    for (i=id, sum[id]=0.0;i< num_steps; i=i+nthreads){
        x = (i+0.5)*step;
        sum[id] += 4.0/(1.0+x*x);
    }
}

for(i=0, pi=0.0;i<nthreads;i++) pi += sum[i] * step;
}
```

Promote scalar to an array dimensioned by number of threads to avoid race condition.

You can't assume that you'll get the number of threads you requested.

Prevent write conflicts with the single.

Performance may suffer due to false sharing of the sum array.

OpenMP recap:

Synchronization (critical region)

```
#include <omp.h>
static long num_steps = 100000;    double step;
#define NUM_THREADS 2
void main ()
{
    int i, id, nthreads; double x, pi, sum;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (i, id, x, sum)
{
    id = omp_get_thread_num();
#pragma omp single
    nthreads = omp_get_num_threads();
    for (i=id, sum=0.0; i< num_steps; i=i+nthreads){
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
#pragma omp critical
    pi += sum * step;
}
}
```

No array, so
no false
sharing.

Note: this method of
combining partial sums
doesn't scale very well.

OpenMP PI Code Recap :

Parallel for with a reduction

```
#include <omp.h>
static long num_steps = 100000;      double step;
#define NUM_THREADS 2
void main ()
{
    int i;  double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=0; i<= num_steps; i++) {
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

For good OpenMP implementations, reduction is more scalable than critical.

i private by default

Note: we created a parallel program without changing any code and by adding 4 simple lines!

OpenMP recap :

Use environment variables to set number of threads

```
#include <omp.h>
static long num_steps = 100000;      double step;
void main ()
{
    int i;  double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel for private(x) reduction(+:sum)
    for (i=0;i<= num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

In practice, you set number of threads by setting the environment variable, OMP_NUM_THREADS

MPI: Pi program

```
#include <mpi.h>
static long num_steps = 100000;
void main (int argc, char *argv[])
{
    int i, my_id, numprocs; double x, pi, step, sum = 0.0 ;
    step = 1.0/(double) num_steps ;
    MPI_Init(&argc, &argv) ;
    MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
    MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
    for (i=my_id; i<num_steps ; i+=numprocs)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    sum *= step ;
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD) ;
}
```

POSIX Threads, Pi Calculation

```
#include <stdlib.h>
#include <sys/time.h>
...

void * compute_pi(void *dat)
{
    int threadid = ((thr_data_t*)dat)->threadid;
    int num_threads = ((thr_data_t*)dat)->num_threads;
    int num_steps = ((thr_data_t*)dat)->num_steps;
    pthread_mutex_t *mtx = ((thr_data_t*)dat)->mtx;
    double *sump = ((thr_data_t*)dat)->sump;
    int i;
    double step;
    double x, local_sum;

    step = 1.0 / num_steps;

    local_sum = 0.0;
    /* round robin distribution of iterations */
    for (i = threadid; i < num_steps; i += num_threads) {
        x = (i - 0.5)*step;
        local_sum += 4.0 / (1.0 + x*x);
    }

    pthread_mutex_lock(mtx);
    *sump = *sump + local_sum;
    pthread_mutex_unlock(mtx);
    return NULL;
}
```

```
int main(int argc, char **argv)
{
    ...

    /* start pi calculation */
    threads = malloc(num_threads * sizeof *threads);
    step = 1.0 / num_steps;
    pthread_mutex_init(&mtx, NULL);

    /* spawn threads to work on computing pi */
    for (i = 0; i < num_threads; i++) {
        dat[i].threadid = i;
        dat[i].num_threads = num_threads;
        dat[i].num_steps = num_steps;
        dat[i].mtx = &mtx;
        dat[i].sump = &sum;
        pthread_create(&threads[i], NULL, compute_pi,
                      (void *)&dat[i]);
    }

    /* join threads */
    for (i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }

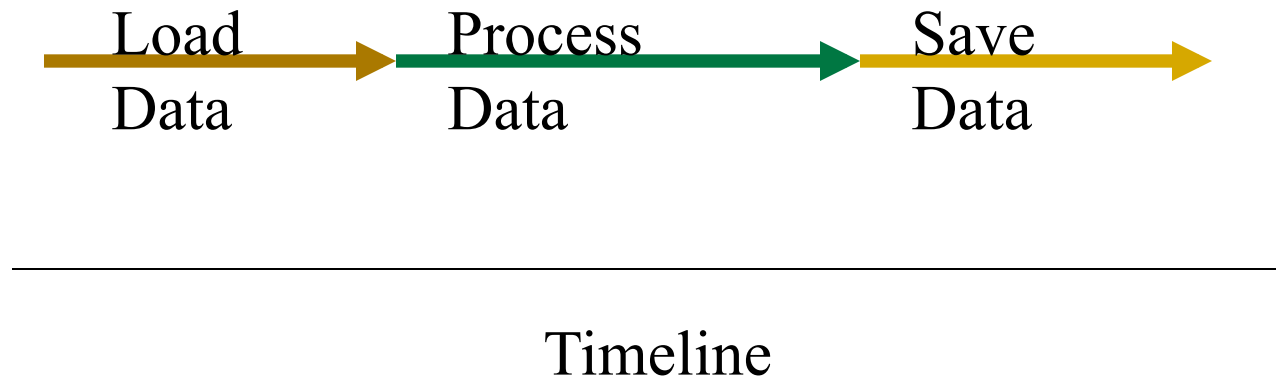
    pi = step * sum;
    free(dat);
    pthread_mutex_destroy(&mtx);
    free(threads);

    ...
}
```

Requires explicit thread/data management

Example: Seismic Data Processing (SDP)

```
for (int iLineIndex=nStartLine; iLineIndex <= nEndLine; iLineIndex++)  
{  
    Loadline(iLineIndex,...);  
    for(j=0;j<iNumTraces;j++)  
        for(k=0;k<iNumSamples;k++)  
            processing();  
    SaveLine(iLineIndex);  
}
```

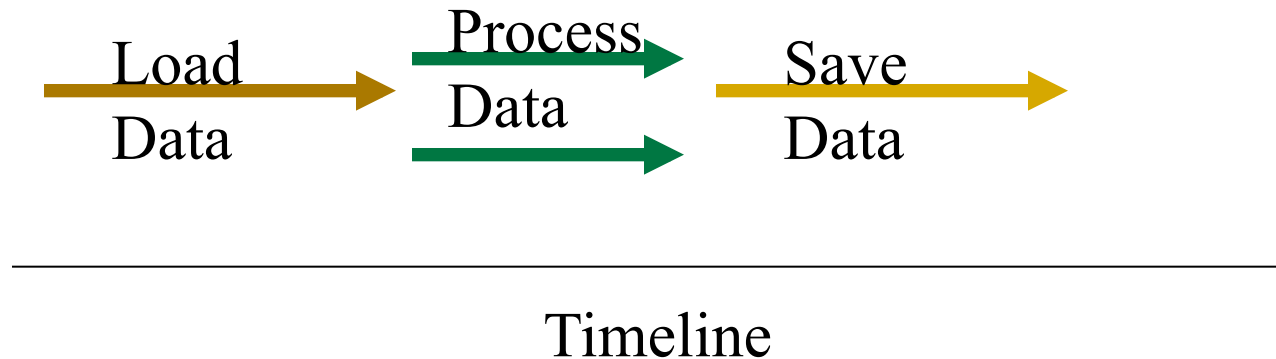


First OpenMP Version of SDP Code

```
for (int iLineIndex=nStartLine; iLineIndex <= nEndLine; iLineIndex++)  
{  
    Loadline(iLineIndex,...);  
    #pragma omp parallel for  
    for(j=0;j<iNumTraces;j++)  
        for(k=0;k<iNumSamples;k++)  
            processing();  
    SaveLine(iLineIndex);  
}
```

Overhead for
entering and leaving
the parallel region

Better performance, but
not too encouraging



Example: Overlap I/O, Processing

```
#pragma omp parallel
#pragma omp sections

{
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) read_input(i);
            (void) signal_read(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_read(i);
            (void) process_data(i);
            (void) signal_processed(i);
        }
    }
    #pragma omp section
    {
        for (int i=0; i<N; i++) {
            (void) wait_processed(i);
            (void) write_output(i);
        }
    }
} /*-- End of parallel sections --*/
```

Input Thread

Processing Thread

Output Thread

Another OpenMP Version of SDP Code

```
Loadline(nStartLine,...); // preload the first line of data
```

```
#pragma omp parallel
```

```
{  
  for (int iLineIndex=nStartLine; iLineIndex <= nEndLine; iLineIndex++)
```

```
{  
  #pragma omp single nowait
```

```
{// loading the next line data, NO WAIT!  
  Loadline(iLineIndex+1,...);
```

```
}  
  #pragma omp for schedule(dynamic)
```

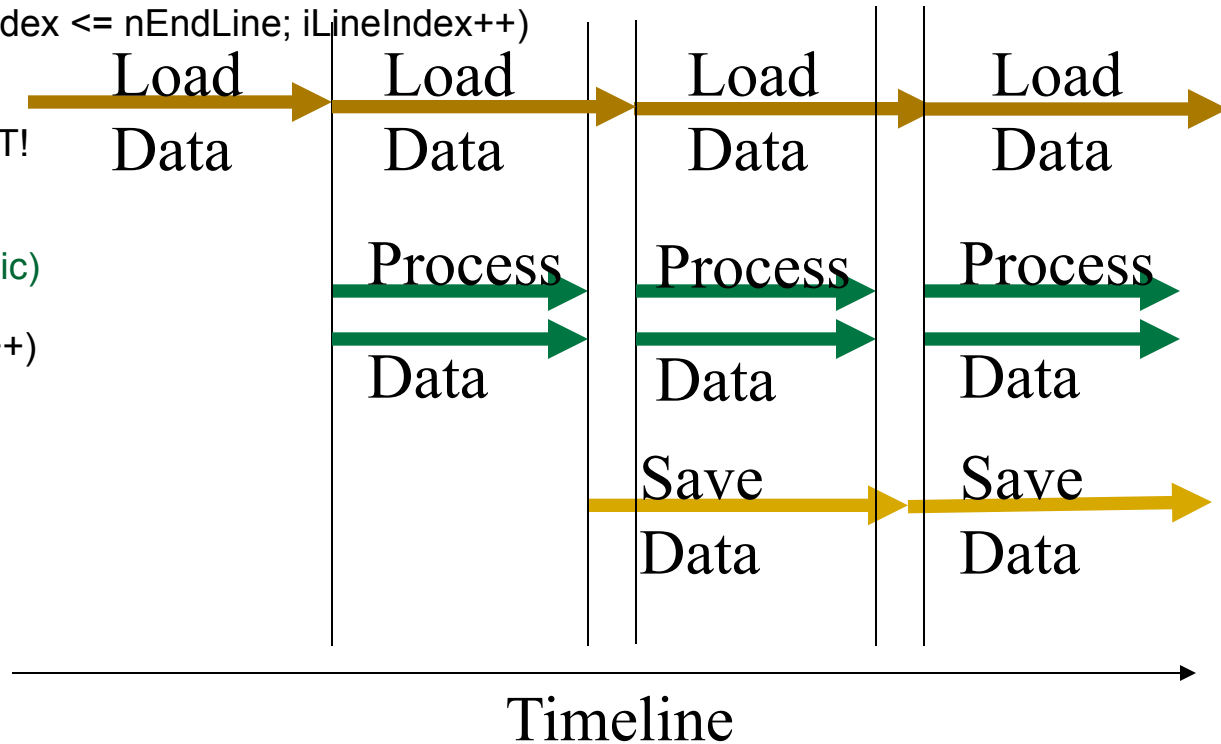
```
  for(j=0;j<iNumTraces;j++)  
    for(k=0;k<iNumSamples;k++)  
      processing();
```

```
#pragma omp single nowait
```

```
{  
  SaveLine(iLineIndex);
```

```
}
```

```
}  
}
```



All 41 examples are available on line!

As well as a forum on <http://www.openmp.org>



THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING



Subscribe to the News Feed

- » » OpenMP Specifications
- » About OpenMP
- » Compilers
- » Resources
- » Discussion

Events

- » IWOMP 2011 (pdf) - 7th International Workshop on OpenMP, 13 - 15, 2011

Input Register

Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.

» webmaster@openmp.org

Search OpenMP.org

Google™ Custom Search

Search

Download Book Examples and Discuss

Ruud van der Pas, one of the authors of the book *Using OpenMP - - Portable Shared Memory Parallel Programming* by Chapman, Jost, and van der Pas, has made 41 of the examples in the book available for download and your use.

These source examples are available as a free download » [here](#) (a zip file) under the BSD license. Each source comes with a copy of the license. Please do not remove this.

Download the examples and discuss in forum:
<http://www.openmp.org/wp/2009/04/download-book-examples-and-discuss>

To make things easier, each source directory has a make file called "Makefile". This file can be used to build and run the examples in the specific directory. Before you do so, you need to activate the appropriate include line in file Makefile. There are include files for several compilers and Unix based Operating Systems (Linux, Solaris and Mac OS to precise).

These files have been put together on a best effort basis. The User's Guide that is bundled with the examples explains this in more detail.

Also, we have created a new forum, » [Using OpenMP - The Book and Examples](#), for discussion and feedback.

Posted on April 2, 2009

Get

- » OpenMP specs

Use

- » OpenMP Compilers

Learn

- » *Using OpenMP* - the book
- » *Using OpenMP* - the examples
- » *Using OpenMP* - the forum
- » Wikipedia
- » OpenMP Tutorial
- » More Resources

Discuss

Summary and Outlook

- ❑ We have seen features of OpenMP and some examples of their use
 - ❑ Powerful, flexible, portable API
 - ❑ Worksharing, synchronization; runtime routines for dynamic threadcount, nesting, ..
- ❑ What is coming?
 - ❑ The latest features in OpenMP 4.0
 - ❑ Using OpenMP in conjunction with MPI
 - ❑ More about optimization
 - ❑ Practical experience